Fixed-point partial recursion in Coq

joint work with Yves Bertot

Vladimir Komendantsky

TYPES workshop
29 March 2008

Motivation: proving program specifications

- A computational limitation of pure CIC is that only structurally recursive definitions are allowed there. This requirement is maintained for the consistency of the logic of CIC, and for the decidability of type-checking.
- Standard programing languages usually impose no restrictions on recursive programs. Many algorithms are general recursive.
- Modelling general recursion in Coq can be employed when proving properties of recursive programs.

Partial recursion

A function is partial recursive if it is

- primitive recursive or
- can be defined from partial recursive functions by means of applications of the minimisation operator.

partial recursion = primitive recursion + minimisation
general recursion = partial recursion

Related work

- Isabelle/HOLCF
- Bertot 2002: `Recursive Definition`
- Paulin-Mohring 2007: formalisation of domain theory with preorders
- Bove 2002: general recursion using ad-hoc predicates
- Capretta 2005: general recursion using coinductive types

The $3n+1$ problem / Collatz conjecture

$$f\ n = \begin{cases} 3n+1 & \text{if } n \text{ is odd} \\ n/2 & \text{if } n \text{ is even} \end{cases}$$

Define a sequence $\{a_i\}$ as follows:

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f\ a_{i-1} & \text{for } i > 0 \end{cases}$$

Collatz conjecture

For any n, there exists an i such that

$$\overbrace{f \cdots f}^{i}\ n = 1$$

The Syracuse function

The Collatz conjecture is equivalent to saying that the function satisfying the following equation terminates for every positive input:

$$f\ n = \begin{cases} 0 & \text{if } n = 1 \\ \mathit{otherwise} \begin{cases} 1 + f(3n+1) & \text{if } n \text{ is odd} \\ 1 + f(n/2) & \text{if } n \text{ is even} \end{cases} \end{cases}$$

The Syracuse function using ad-hoc predicates

```
Definition odd_sum : forall x, {odd x = true}+{odd x = false}.

Inductive sdom : nat -> Prop :=
  sd1 : sdom 1
| sd2 : forall x, x <> 1 -> odd x =false -> sdom (div2 x) -> sdom x
| sd3 : forall x, x <> 1 -> odd x =true -> sdom (3*x+1) -> sdom x.

Lemma sdom_inv1 : forall x, sdom x -> x <> 1 -> odd x = true -> sdom (3*x+1).
Lemma sdom_inv2 : forall x, sdom x -> x <> 1 -> odd x = false -> sdom (div2 x).
Lemma sdom_inv3 : forall x, x = 0 -> ~sdom x.

Lemma ssn1 : forall x p, x = S (S p) -> x <> 1.

Fixpoint syracuse (x:nat)(h:sdom x) {struct h} : nat :=
  match x as n return x = n -> nat with
    0 => fun q => match sdom_inv3 x q h return nat with end
  | 1 => fun q => 0
  | S (S p) => fun q =>
    match odd_sum x with
      left o => 1+(syracuse (3*x+1) (sdom_inv1 x h (ssn1 x p q) o))
    | right e => 1+(syracuse (div2 x) (sdom_inv2 x h (ssn1 x p q) e))
    end
  end (refl_equal x).
```

The Syracuse function, continued in Coq

```
Definition eq1 (n:nat) : bool := match n with 1 => true | _ => false end.

Definition bind (A B:Type)(v:option A)(f:A->option B) : option B :=
  match v with Some a => f a | None => None end.

Fixpoint syracuse (x:nat) : option nat :=
  if eq1 x then Some 0
  else if odd x then bind (syracuse (3*x+1)) (fun v => Some (S v))
    else bind (syracuse (div2 x)) (fun v => Some (S v)).
```

This attempt yields an error message:

```
Error:
Recursive definition of syracuse is ill-formed.
In environment
syracuse : nat -> option nat
x : nat
Recursive call to syracuse has principal argument equal to
"3 * x + 1"
instead of a subterm of x.
```

A new command

```
 A_New_Command syracuse (x:nat) :  option nat :=
  if eq1 x then Some 0
  else if odd x then bind (syracuse (3*x+1)) (fun v =>
Some (S v))
    else bind (syracuse (div2 x)) (fun v => Some (S v)).
```

A new command

```
A_New_Command syracuse (x:nat) :  option nat :=
  if eq1 x then Some 0
  else if odd x then bind (syracuse (3*x+1)) (fun v =>
Some (S v))
    else bind (syracuse (div2 x)) (fun v => Some (S v)).
```

Our method: define a functional

```
Definition f_syracuse (syracuse:nat->option nat) :
nat->option nat :=
  fun n =>
    if eq1 x then Some 0
    else if odd x then bind (syracuse (3*x+1)) (fun v =>
Some (S v))
      else bind (syracuse (div2 x)) (fun v => Some (S v)).
```

and find its least fixed point

```
Definition syracuse := fixp f_syracuse.
```

Denotational semantics of a programming language: the while operator

```
Set Implicit Arguments.
Unset Strict Implicit.

Definition ifthenelse (A:Type)(t:option bool)(v w: option A) :=
  match t with
  | Some true => v
  | Some false => w
  | None => None
  end.

Definition bind (A B:Type)(v:option A)(f:A->option B) : option B :=
  match v with Some a => f a | None => None end.
```

```
Definition f_while (A:Type)(t:A->option bool)
     (f g:A->option A): A->option A :=
  fun a:A => ifthenelse (t a) (bind (f a) g) (Some a).
```

```
Definition while := fun t f => fixp (f_while t f).
```

The Knaster–Tarski fixed point theorem

Theorem (complete preorder version of the Knaster–Tarski)

Given a monotonic function f on a complete preorder, consider the following transfinite sequence:

$$
\begin{aligned}
x_o &= \bot \\
x_{\alpha+1} &= f(x_\alpha) \\
x_\beta &= \text{ the least upper bound of the chain } \{f(x_\alpha)\}_{\alpha<\beta} \\
&\quad \text{if } \beta \text{ is a limit ordinal.}
\end{aligned}
$$

The function f has a least fixed point. Moreover, if f is continuous then the least fixed point is obtained in at most ω iterations.

Iteration and the least fixed point operator

Fixpoint f_iter $(f : D \xrightarrow{m} D)(n : \mathsf{nat}_{\mathsf{ord}}) : D :=$
match n with
  $0 \Rightarrow \bot$
  $\mid \mathsf{S}\ n' \Rightarrow f(\mathsf{f\_iter}\ n')$
end.

Iteration and the least fixed point operator

Fixpoint f_iter $(f : D \xrightarrow{m} D)(n : \mathsf{nat}_{\mathsf{ord}}) : D :=$
match n with
  $0 \Rightarrow \bot$
  $\mid$ S $n' \Rightarrow f(\text{f\_iter } n')$
end.
Definition iter $(f : D \xrightarrow{m} D)$: chain D

Iteration and the least fixed point operator

Fixpoint f_iter $(f : D \xrightarrow{m} D)(n : \mathsf{nat}_{\mathsf{ord}}) : D :=$
match n with
  $0 \Rightarrow \bot$
  $\mid$ S $n' \Rightarrow f(\text{f\_iter } n')$
end.
Definition iter $(f : D \xrightarrow{m} D)$: chain D
Definition f_fixp $(f : D \xrightarrow{m} D) : D :=$ lub (iter f).

Iteration and the least fixed point operator

Fixpoint f_iter $(f : D \xrightarrow{m} D)(n : \mathrm{nat}_{\mathrm{ord}}) : D :=$
match n with
  $0 \Rightarrow \bot$
  $\mid$ S $n' \Rightarrow f$(f_iter n')
end.
Definition iter $(f : D \xrightarrow{m} D)$: chain D
Definition f_fixp $(f : D \xrightarrow{m} D) : D :=$ lub (iter f).

Under the continuity assumption on f, the required fixed point equation holds:

Lemma f_fixp_eq : (f_fixp f) $==$ f (f_fixp f).

The least fixed point operator
Definition fixp (D:cpo) : $(D \xrightarrow{C} D) \xrightarrow{C} D$.

A Coq formalisation: Kleene's fixed point theorem

In the setting of preorders, Kleene's fixpoint theorem can be summarised as follows:

> *In a complete preorder, every continuous function has a least fixed point for the derived equality.*

Theorem (formal Coq statement of Kleene's fixpoint theorem)

$$\forall (D : cpo)(f : D \stackrel{C}{\rightarrow} D),$$
$$f(\mathit{fixp}\ f) == \mathit{fixp}\ f \wedge (\forall x, f\ x \leq x \rightarrow \mathit{fixp}\ f \leq x).$$

Flat preorders

Inductive option $(A : \text{Type}) : \text{Type} :=$

$$\text{Some} : A \rightarrow \text{option A} \mid \text{None} : \text{option } A.$$

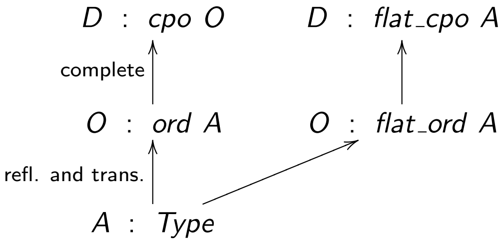
The flat preorder on a type A is specified by

$$\leq_{\text{option } A}$$

such that, for $x\ y : A$,

$$x \leq_{\text{option } A} y \quad \text{iff} \quad x = y \text{ or } x = \text{None}$$

Hierarchy of structures



Classical postulates

We rely on the Classical and ClassicalDescription extensions of the Coq libraries. We only make use of two new axioms:

- excluded middle,
- constructive definite description.

Constructive definite description

$\forall (A : \mathsf{Type})(P : A \to \mathsf{Prop}), (\exists !\ x : A, P\ x) \to \{x : A \mid P\ x\}.$

Implementation remark

Both axioms are used conservatively only in the certificate which has no effect on the behaviour of recursive functions being defined, and is not extracted to a program.

Example: Minimisation

The minimisation functional μ

For all A : Type, $f : A \times \text{nat} \rightarrow \text{nat}$, the value of μf is a function $g : A \rightarrow \text{nat}$ such that

$$g\ x = \begin{cases} y \text{ if } y \text{ is the least value s.t. } f(x, y) = 0 \text{ holds} \\ \text{undefined, otherwise} \end{cases}$$

Minimisation in Coq

```
Variable A : Type.
Variable f : A -> nat -> option nat.

Definition f_mu (mu : A -> nat -> option nat) :
    A -> nat -> option nat :=
  fun x y =>
    match f x y with
      None => None
    | Some 0 => Some y
    | _ => mu x (S y)
    end.

Lemma f_mu_monotonic :
  @monotonic (A -o-> nat -o-> &ord nat) (A -o-> nat -o-> &ord nat) f_mu.
Definition mono_mu :
  (A -o-> nat -o-> &ord nat) -m-> (A -o-> nat -o-> &ord nat).
Lemma mono_mu_continuous : continuous mono_mu.
Definition cont_mu :
  (A -O-> nat -O-> &cpo nat) -C-> (A -O-> nat -O-> &cpo nat).

Definition mu := fixp cont_mu.
```

The Minimisation example using the new command Fcpo Function

```
Variable A : Type.
Variable f : A -> nat -> option nat.

Fcpo Function mu : A -> nat -> option nat :=
  fun x y =>
    match f x y with
      None => None
    | Some 0 => Some y
    | _ => mu x (S y)
    end.
```

Then the system asks to manually specify two functions with the following types:

```
(A -o-> nat -o-> &ord nat) -m-> (A -o-> nat -o-> &ord nat)
(A -O-> nat -O-> &cpo nat) -C-> (A -O-> nat -O-> &cpo nat)
```

Next, the functional is defined automatically.

Example: Minimisation – continued

$\lambda xy.|x - y^2|$

```
Definition abs_x_minus_y_squared (x y :  nat) :=
  Some ((x - y*y) + (y*y - x)).
```

The value of $\mu(\lambda xy.|x - y^2|)k$ is defined if and only if k is a perfect square:

```
Definition perfect_sqrt (x:nat) :=
  mu abs_x_minus_y_squared x 0.

Lemma perfect_sqrt_None : forall x:nat,
  (forall y:nat, ~ x = y*y) -> perfect_sqrt x = None.
```

Extraction constants

fix f = f (fix f)

Extraction constants

fix f = f (fix f)

let rec fix f = f (fix f)

Extraction constants

fix f = f (fix f)

let rec fix f = f (fix f)

let rec fix f = f (fun y $\rightarrow$ fix f y)

Extraction constants

fix f = f (fix f)

let rec fix f = f (fix f)

let rec fix f = f (fun y → fix f y)

```
Extract Constant fixp =>
  "let rec t d f x = f (fun y -> t d f y) x in t".

Extract Constant constructive_definite_description =>
  "Obj.magic".
```

Extraction of a functional program from the definition of the partial function perfect_sqrt

```
let mu f =
  Obj.magic (Obj.magic (fixp (funcpo (funcpo flat_cpo))))
    (Obj.magic
      (Obj.magic
        (Obj.magic (fun x x0 x1 ->
          match f x0 x1 with
            | Some n ->
                (match n with
                   | O -> Some x1
                   | S n0 -> x x0 (S x1))
            | None -> None))))

let abs_x_minus_y_squared x y =
  Some (plus (minus x (mult y y)) (minus (mult y y) x))

let perfect_sqrt x =
  Obj.magic mu abs_x_minus_y_squared x O
```

Work in progress

- implementation in Coq of the new command Fcpo Function
- automation of routine monotonicity and continuity proofs
- optimisation of the extracted code (removing the option type and occurrences of Obj.magic)
- providing additional tools, e.g., for proofs by fixed point induction

Complete preorders

A complete preorder is a record consisting of

- a preorder O,
- an element $\bot : O$ and
- a least upper bound function $\text{lub} : (\text{chain } O) \to O$ satisfying the three laws below:

$$\forall x : O, \bot \leq x$$

$$\forall (c : \text{chain } O)(n : \text{nat}_{\text{ord}}), c\ n \leq \text{lub } c$$

$$\forall (c : \text{chain } O)(x : O), (\forall n : \text{nat}_{\text{ord}}, c\ n \leq x) \to \text{lub } c \leq x$$

Continuous functions

For cpos D_1 and D_2, a monotonic function $f : D_1 \rightarrow D_2$ is continuous whenever, for any chain c on D_1,

$$f(\mathsf{lub}\ c) \leq \mathsf{lub}(f \circ c)\ .$$